# Assessed coursework: Integrating R and C++

Cecina Babich Morrow

2024-02-09

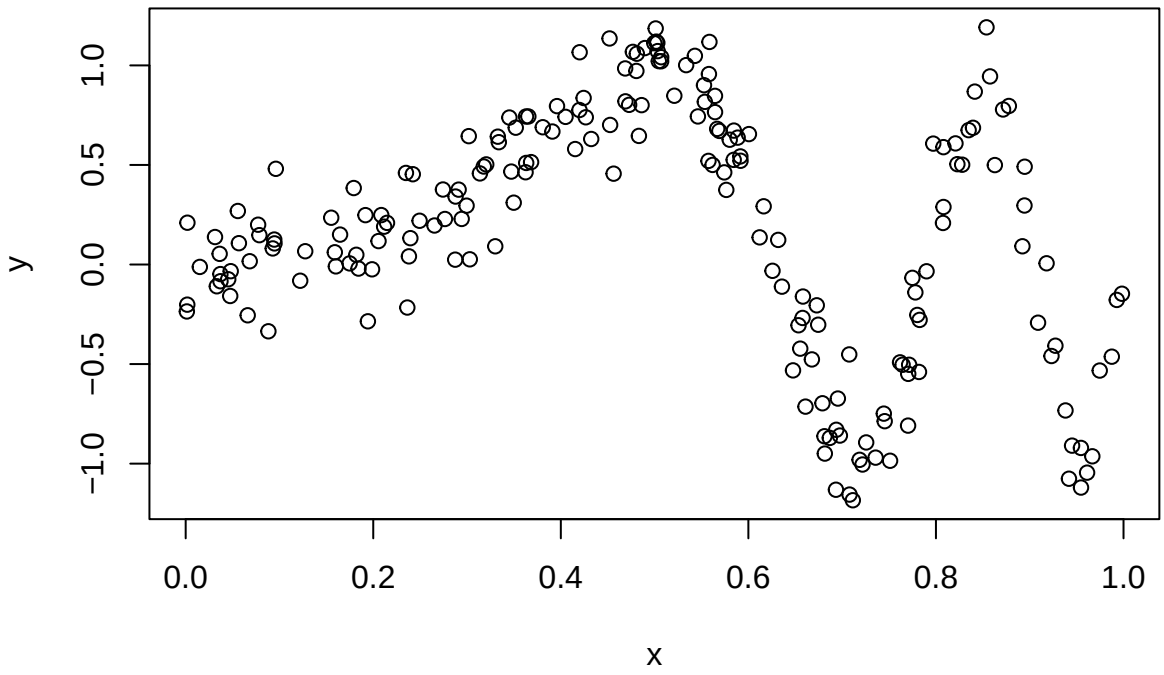
## Adaptive kernel regression smoothing

We will consider data generated from the model

$$y_i = \sin(\alpha \pi x^3) + z_i$$

where $z_i \sim N(0, \sigma^2)$, $i \in \{1, ..., n\}$.

```
set.seed(998)
# n = 200
nobs <- 200
x <- runif(nobs)
# alpha = 4, sigma = 0.2
y <- sin(4*pi*x^3) + rnorm(nobs, 0, 0.2)
plot(x, y)
```



We want to model this data using a kernel regression smoother (KRS) by estimating $\mu(x) = \mathbb{E}(y|x)$. The KRS estimator is given by

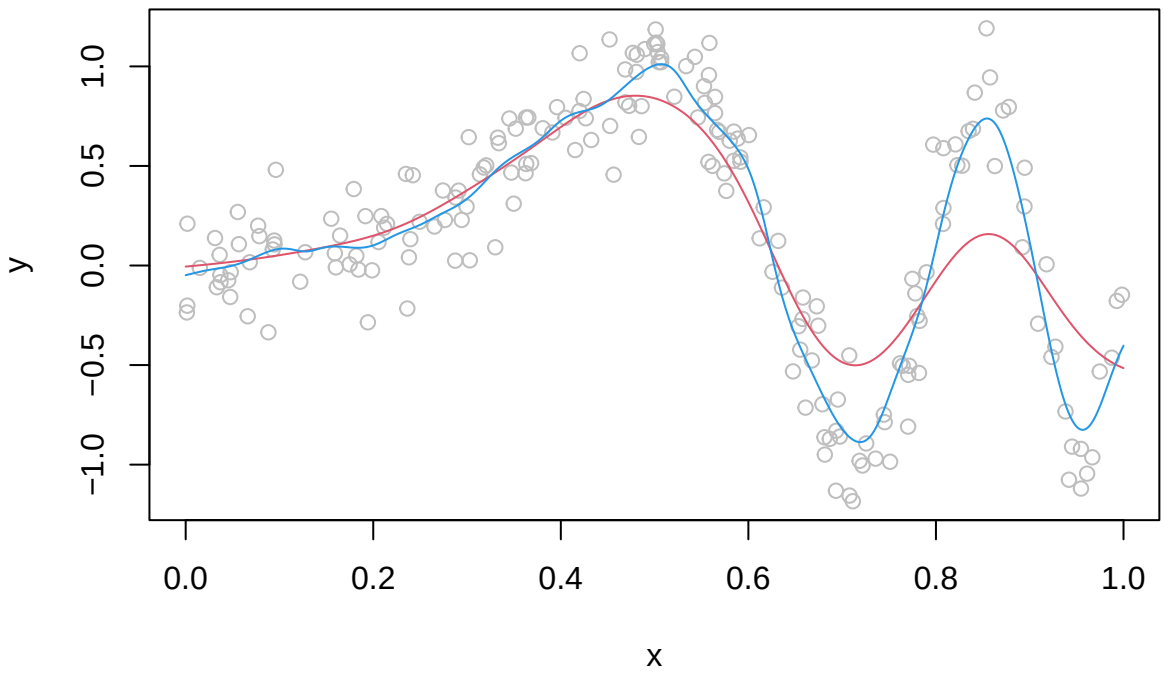
$$\hat{\mu}(x) = \frac{\sum_{i=1}^{n} \kappa_\lambda(x, x_i) y_i}{\sum_{i=1}^{n} \kappa_\lambda(x, x_i)}$$

where $\kappa$ is a kernel function with bandwidth $\lambda > 0$. The following R function uses the Gaussian kernel function with variance $\lambda^2$:

```
meanKRS <- function(y, x, x0, lam){

 n <- length(x)
 n0 <- length(x0)

 out <- numeric(n0)
 for(ii in 1:n0){
  out[ii] <- sum( dnorm(x, x0[ii], lam) * y ) / sum( dnorm(x, x0[ii], lam) )
 }

 return( out )
}
```

We can compare the performance of the KRS estimator with different bandwidths:

```
xseq <- seq(0, 1, length.out = 1000)
muSmoothLarge <- meanKRS(y = y, x = x, x0 = xseq, lam = 0.06)
muSmoothSmall <- meanKRS(y = y, x = x, x0 = xseq, lam = 0.02)
plot(x, y, col = "grey")
lines(xseq, muSmoothLarge, col = 2)
lines(xseq, muSmoothSmall, col = 4)
```




**Q1a**

We want to write a C++ version of the `meanKRS` function. The function is as follows (available on GitHub here:

```
#include <R.h>
#include <Rinternals.h>
#include <Rmath.h>

SEXP meanKRS_C(SEXP y_vec, SEXP x_vec, SEXP x0_vec, SEXP lambda_param)
{

  // Get vector lengths from R object
  int n = length(x_vec);
  int n0 = length(x0_vec);
  // Set up and protect output vector
  SEXP out = PROTECT(allocVector(REALSXP, n0));

  // Get pointers to the R object input parameters
  double *y = REAL(coerceVector(y_vec, REALSXP));
  double *x = REAL(coerceVector(x_vec, REALSXP));
  double *x0 = REAL(coerceVector(x0_vec, REALSXP));
  double lambda = REAL(lambda_param)[0];

  // Calculate the sum of the results of dnorm
```

```
  for (int i = 0; i < n0; i++)
  {
    double sum_dens_norm_y = 0;
    double sum_dens_norm = 0;

    for (int j = 0; j < n; j++)
    {
      double dens_norm = dnorm(x[j], x0[i], lambda, 0);
      sum_dens_norm_y += dens_norm * y[j]; // Sum of dnorm * y
      sum_dens_norm += dens_norm; // Sum of dnorm
    }

    // Result is ratio
    REAL(out)[i] = sum_dens_norm_y / sum_dens_norm;
  }

  UNPROTECT(1);

  return out;
}
```

We can compile the code and then load the function as follows:

```
# Compile the code
system(paste0("R CMD SHLIB ", here("portfolios/02_interfacing_r_with_c++/meanKRS_C.c")))
# Load the binary code
dyn.load(here("portfolios/02_interfacing_r_with_c++/meanKRS_C.so"))
# Check if the function is loaded
is.loaded("meanKRS_C")
```

```
## [1] TRUE
```

Next, we can call it using `.Call` and compare the results with the R function:

```
c_smooth_test <- .Call("meanKRS_C",
                       y_vec = y,
                       x_vec = x,
                       x0_vec = xseq,
                       lambda_param = 0.06)

# Compare with results of R function
all.equal(muSmoothLarge, c_smooth_test)
```

```
## [1] TRUE
```

```
# Compare computing time
krs_R <- function() meanKRS(y = y, x = x, x0 = xseq, lam = 0.06)
krs_C <- function() .Call("meanKRS_C", y = y, x = x, x0 = xseq, lambda = 0.06)
library(microbenchmark)
microbenchmark(krs_R(), krs_C(), times = 500)
```

```
## Unit: milliseconds
##     expr      min       lq      mean   median        uq       max neval
##  krs_R() 9.395225 9.893374 10.227024 9.974485 10.107897 14.514225   500
##  krs_C() 2.678687 2.850017  2.908353 2.888659  2.941629  3.640658   500
```

The C version of the function is over 3.5 times faster than the R version on average.

### Q1b

We now want to implement $k$-fold cross-validation for selecting the bandwidth $\lambda$. We can first do so in R:

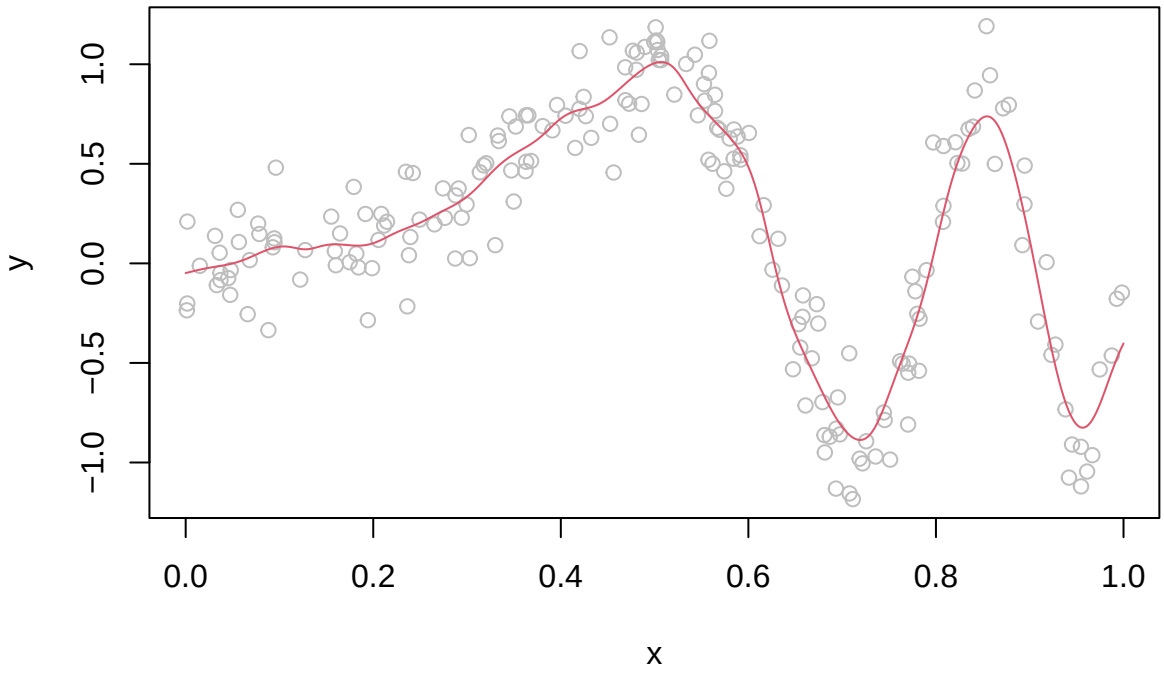
```
krsCV <- function(y, x, k, lam_seq){

  n <- length(x)
  groups <- sample(rep(1:k, length.out = n), size = n)

  mse_table <- data.frame(lambda = rep(lam_seq, each = k),
                          fold = rep(1:k, length(lam_seq)),
                          mse = NA)
  for (lambda in lam_seq) {
    for (i in 1:k) {
      # Set up training and testing sets
      x_train <- x[groups != i]
      y_train <- y[groups != i]
      x_test <- x[groups == i]
      y_test <- y[groups == i]

      # Fit the model on the training set and get values for x_test
      mu_pred <- meanKRS(y = y_train, x = x_train, x0 = x_test, lam = lambda)

      # Calculate MSE on the testing set
      mse <- mean((mu_pred - y_test)^2)

      mse_table$mse[mse_table$lambda == lambda & mse_table$fold == i] <- mse
    }

  }

  mean_mse <- mse_table %>%
    group_by(lambda) %>%
    summarise(mean_mse = mean(mse))

  best_lambda <- mean_mse$lambda[which.min(mean_mse$mean_mse)]

  return(best_lambda)
}
```

Using this R function, we can find the value of $\lambda$ with the lowest average mean squared error (MSE) over 5-fold cross-validation:

```
best_lambda <- krsCV(y = y, x = x, k = 5, lam_seq = seq(0.01, 0.1, by = 0.01))
# See the best lambda value from 5-fold CV
best_lambda
```

```
## [1] 0.02
```

```
xseq <- seq(0, 1, length.out = 1000)
mu_best_lambda <- meanKRS(y = y, x = x, x0 = xseq, lam = best_lambda)
plot(x, y, col = "grey")
lines(xseq, mu_best_lambda, col = 2)
```




We can see the fit above using $\lambda = 0.02$.

Next, we can write an equivalent function in C++ to compare results and performance. The `krsCV_Cpp.cpp` file referenced here is available on GitHub here.

```
# dyn.unload(here("portfolios/02_interfacing_r_with_c++/krsCV_Cpp.so"))
# Compile the code
system(paste0("R CMD SHLIB ", here("portfolios/02_interfacing_r_with_c++/krsCV_Cpp.cpp")))
# Load the binary code
dyn.load(here("portfolios/02_interfacing_r_with_c++/krsCV_Cpp.so"))
# Check if the function is loaded
is.loaded("krsCV_Cpp")
```

```
## [1] TRUE
```

We can now use the C++ function to perform 5-fold cross-validation and return the value of $\lambda$ yielding the highest average MSE across the 5 folds.

```r
# Get the best lambda value from 5-fold CV
# Using the C++ function
c_best_lambda <- .Call("krsCV_Cpp",
                       y_vec = y,
                       x_vec = x,
                       k_val = as.integer(5),
                       lambda_sequence = seq(0.01, 0.1, by = 0.01))
c_best_lambda
```

```
## [1] 0.02
```

We can compare the computational performance of `krsCV_Cpp` to our R function `krsCV`:

```r
# Compare computing time
krsCV_R <- function() krsCV(y = y,
                            x = x,
                            k = 5,
                            lam_seq = seq(0.01, 0.1, by = 0.01))
krsCV_Cpp <- function() .Call("krsCV_Cpp",
                       y_vec = y,
                       x_vec = x,
                       k_val = as.integer(5),
                       lambda_sequence = seq(0.01, 0.1, by = 0.01))

microbenchmark(krsCV_R(), krsCV_Cpp(), times = 500)
```

```
## Unit: milliseconds
##         expr       min        lq      mean    median        uq       max neval
##    krsCV_R() 18.837064 19.798850 21.064756 20.944796 21.620483 27.134208   500
##  krsCV_Cpp()  4.750247  5.062461  5.332995  5.374989  5.544749  8.929007   500
```

Once again, the C++ version is much faster than the R version, this time around 4 times faster.

### Q2

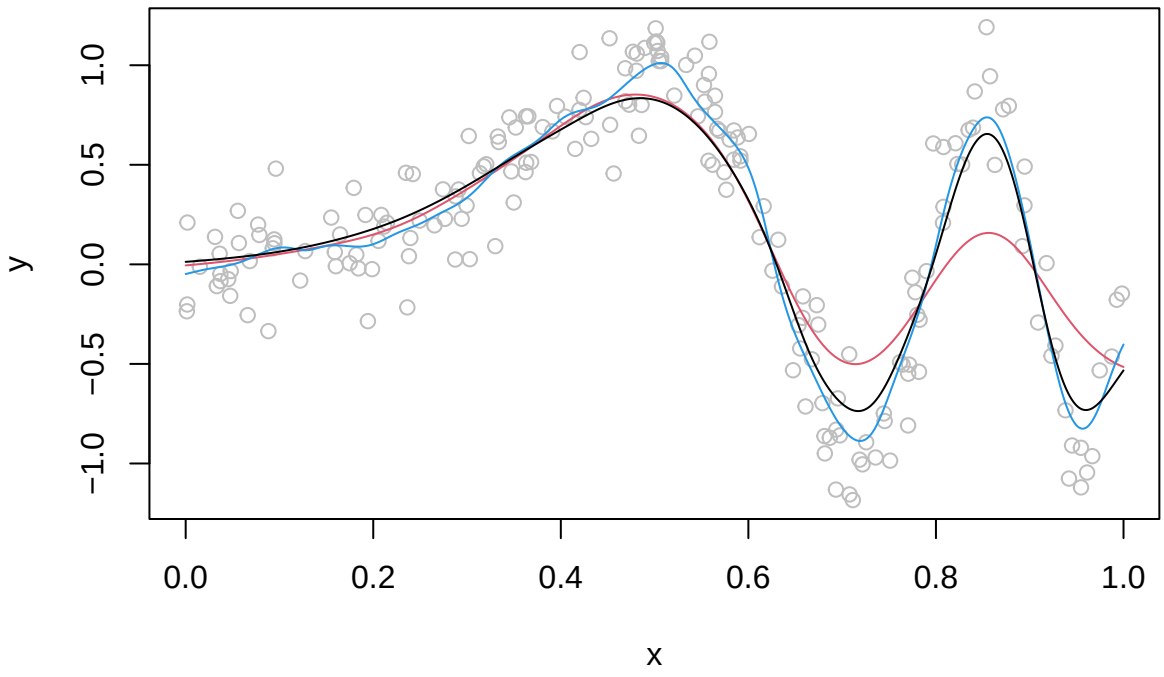
The following R function allows the bandwidth to depend on $x$, i.e. $\lambda = \lambda(x)$:

```r
mean_var_KRS <- function(y, x, x0, lam){

 n <- length(x)
 n0 <- length(x0)
 mu <- res <- numeric(n)

 out <- madHat <- numeric(n0)

 for(ii in 1:n){
  mu[ii] <- sum( dnorm(x, x[ii], lam) * y ) / sum( dnorm(x, x[ii], lam) )
 }

 resAbs <- abs(y - mu)
 for(ii in 1:n0){
```

```
 madHat[ii] <- sum( dnorm(x, x0[ii], lam) * resAbs ) / sum( dnorm(x, x0[ii], lam) )
}

w <- 1 / madHat
w <- w / mean(w)

for(ii in 1:n0){
 out[ii] <- sum( dnorm(x, x0[ii], lam * w[ii]) * y ) /
            sum( dnorm(x, x0[ii], lam * w[ii]) )
}

return( out )
}
```

We can see the results here, where the red and blue lines show the fits from earlier where $\lambda = 0.06$ and $\lambda = 0.02$, respectively, and the black line shows the fit using the adaptive bandwidth function `mean_var_KRS`:

```
xseq <- seq(0, 1, length.out = 1000)
muSmoothAdapt <- mean_var_KRS(y = y, x = x, x0 = xseq, lam = 0.06)
plot(x, y, col = "grey")
lines(xseq, muSmoothLarge, col = 2) # red
lines(xseq, muSmoothSmall, col = 4) # blue
lines(xseq, muSmoothAdapt, col = 1) # black
```




We now want to write a version of `mean_var_KRS` in C++ and compare the results and performance with the R function. Code for the C++ function can be found on GitHub here.

```
# dyn.unload(here("portfolios/02_interfacing_r_with_c++/mean_var_krs_Cpp.so"))
# Compile the code
system(paste0("R CMD SHLIB ",
              here("portfolios/02_interfacing_r_with_c++/mean_var_krs_Cpp.cpp")))
# Load the binary code
dyn.load(here("portfolios/02_interfacing_r_with_c++/mean_var_krs_Cpp.so"))
# Check if the function is loaded
is.loaded("mean_var_krs_Cpp")
```

```
## [1] TRUE
```

We can use the C++ function to fit the model and compare the results with the R function:

```
mean_var_test <- .Call("mean_var_krs_Cpp",
                       y_vec = y,
                       x_vec = x,
                       x0_vec = xseq,
                       lambda_param = 0.06)

# Compare with results of R function
all.equal(muSmoothAdapt, mean_var_test)
```

```
## [1] TRUE
```

```
# Compare computing time
var_krs_R <- function() mean_var_KRS(y = y, x = x, x0 = xseq, lam = 0.06)
var_krs_C <- function() .Call("mean_var_krs_Cpp",
                              y_vec = y,
                              x_vec = x,
                              x0_vec = xseq,
                              lambda_param = 0.06)
microbenchmark(var_krs_R(), var_krs_C(), times = 500)
```

```
## Unit: milliseconds
##         expr       min        lq      mean    median       uq       max neval
##  var_krs_R() 21.539857 23.497310 24.234656 23.854248 24.70017 64.890865   500
##  var_krs_C()  6.267608  6.802672  6.949851  6.930601  7.09589  8.699495   500
```

Once again, the implementation in C++ is much faster than in R, a little over 3.5 times faster.